This message makes little sense to a beginner, so we won't attempt to interpret it right now. However, one thing is obvious; **tput** requires additional input to work properly. To make **tput** work, follow **tput** with the word `clear`:

```
tput clear                              clear is an argument to tput
```

The screen clears and the prompt and cursor are positioned at the top-left corner. Some systems also offer the **clear** command, but the standard UNIX specifications (like POSIX) don't require UNIX systems to offer this command. You must remember to use **tput clear** to clear the screen because we won't be discussing this command again in this text.

Note: The additional word used with **tput** isn't a command, but is referred to as an *argument*. Here, `clear` is an argument to **tput**, and the fact that **tput** refused to work alone indicates that it always requires an argument (sometimes more). And, if `clear` is one argument, there could be others. We'll often refer to the *default* behavior of a command to mean the effect of a using a command without any arguments.

1.4.6 cal: The Calendar

cal is a handy tool that you can invoke any time to see the calendar of any specific month, or a complete year. To see the calendar for the month of July, 2006, provide the month number and year as the two arguments to **cal**:

```
$ cal 7 2006                            Command run with two arguments
     July 2006
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

With **cal**, you can produce the calendar for any month or year between the years 1 and 9999. This should serve our requirements for some time, right? We'll see more of this command later.

1.4.7 who: Who Are the Users?

UNIX is a system that can be concurrently used by multiple users, and you might be interested in knowing the people who are also using the system like you. Use the **who** command:

```
$ who
kumar      console      May  9 09:31    (:0)
vipul      pts/4        May  9 09:31    (:0.0)
raghav     pts/5        May  9 09:32    (saturn.heavens.com)
```

There are currently three users—kumar, vipul and raghav. These are actually the user-ids or usernames they used to log in. The output also includes the username, kumar, which you entered

18058

at the `login:` prompt to gain entry to the system. The second column shows the name of the terminal the user is working on. Just as all users have names, all terminals, disks and printers also have names. You'll see later that these names are represented as *files* in the system. The date and time of login are also shown in the output. Ignore the last column for the time being.

Observe also that the output of **who** doesn't include any headers to indicate what the various columns mean. This is an important feature of the system, and is in some measure responsible for the unfriendly image that UNIX has acquired. After you have completed Chapter 8, you'll discover that it is actually a blessing in disguise.

You logged in with the name kumar, so the system addresses you by this name and associates kumar with whatever work you do. Create a file and the system will make kumar the owner of the file. Execute a program and kumar will be the owner of the *process* (next topic) associated with your program. Send mail to another user and the system will inform the recipient that mail has arrived from kumar.

Note: UNIX isn't just a repository of commands producing informative output. You can extract useful information from command output for using with other commands. For instance, you can extract the day of the week (here, Fri) from the **date** output and then devise a program that does different things depending on the day the program is invoked. You can also "cut" the user-ids from the **who** output and use the list with the **mailx** command to send mail to all users currently logged in. The facility to perform these useful tasks with one or two lines of code makes UNIX truly different from other operating systems.

1.4.8 ps: Viewing Processes

We observed that the shell program is always running at your terminal. Every command that you run gives rise to a *process*, and the shell is a process as well. To view all processes that you are responsible for creating, run the **ps** command:

```
$ ps
    PID TTY        TIME CMD
    364 console  0:00 ksh                                    Shell running all the time!
```

Unlike **who**, **ps** generates a header followed by a line containing the details of the **ksh** process. When you run several programs, there will be multiple lines in the **ps** output. **ksh** represents the Korn shell (an advanced shell from AT&T) and is constantly running at this terminal. This process has a unique number 364 (called the *process-id* or PID), and when you log out, this process is killed.

Note: Even though we are using the Korn shell here, you could be using another shell. Instead of **ksh**, you could see **sh** (the primitive Bourne shell), **csh** (C shell—still popular today) or **bash** (Bash shell—a very powerful shell and recommended for use). Throughout this book, we'll be comparing the features of these shells and discover features that are available in one shell but not in another. If a command doesn't produce output as explained in this text, it can often be attributed to the shell.

1.4.9 ls: Listing Files

Your UNIX system has a large number of files that control its functioning, and users also create files on their own. These files are organized in separate folders called *directories*. You can list the names of the files available in this directory with the **ls** command:

```
$ ls
README                                          Uppercase first
chap01
chap02
chap03
helpdir
progs
```

ls displays a list of six files, three of which actually contain the chapters of this textbook. Note that the files are arranged alphabetically with uppercase having precedence over lower (which we call the **ASCII collating sequence**).

Since the files containing the first three chapters have similar filenames, UNIX lets you use a special short-hand notation (*) to access them:

```
$ ls chap*
chap01
chap02
chap03
```

Sometimes, just displaying a list of filenames isn't enough; you need to know more about these files. For that to happen, **ls** has to be used with an *option*, -l, between the command and filenames:

```
$ ls -l chap*                                            -l is an option
-rw-r--r--   1 kumar    users     5609 Apr 23 09:30 chap01
-rw-r--r--   1 kumar    users    26129 May 14 18:55 chap02
-rw-r--r--   1 kumar    users    37385 May 15 10:30 chap03
```

The argument beginning with a hyphen is known as an *option*. The characteristic feature of most command options is that they begin with a - (hyphen). An option changes the default behavior (i.e. when used without options) of a command, so if ls prints a columnar list of files, the -l option makes it display some of the attributes as well.

1.4.10 Directing Output to a File

UNIX has simple symbols (called *metacharacters*) for creating and storing information in files. Instead of viewing the output of the ls command on the terminal, you can save the information in a file, list, by using a special symbol, > (the right chevron character on your keyboard):

```
$ ls > list
$ _                                      Prompt returns—no display on terminal
```

You see nothing on the terminal except the return of the prompt. The shell is at work here. It has a mechanism of *redirecting* any output, normally coming to the terminal, to a disk file. To check whether the shell has actually done the job, use the **cat** command with the filename as argument:

```
$ cat list                                              cat displays a file's contents
README
chap01
chap02
chap03
helpdir
progs
```

This single line of code lets us catch a glimpse of the UNIX magic. The shell sees the > before runs, so it gets to act first. It then opens the file following the > (here, `list`). **`ls`** runs next and looks up the "table of contents" of the directory to find out the filenames in it. But the shell h already manipulated things in such a way that the **`ls`** output doesn't come to the terminal but the file opened by the shell on its behalf.

Note: You list files in a directory with **`ls`** and display file contents with **`cat`**. You can get more details a file using the -l option with **`ls`**.

1.4.11 wc: Counting Number of Lines in a File

How many lines are there in the file? The **`wc`** command answers this question:

```
$ wc list
     6       6      42 list
```

Observe once again the brevity that typically characterizes UNIX; it merely echoes three numbe along with the filename. You have to have the manual or this text in front of you to know that th file `list` contains 6 lines, 6 words and 42 characters.

1.4.12 Feeding Output of One Command to Another

Now you should see something that is often hailed as the finest feature of the system. Previousl you used **`ls`** to list files, and then the > symbol to save the output in the file `list`. You then counte the number of lines, words and characters in this file with **`wc`**. In this way, you could *indirect* count the number of files in the directory. The shell can do better; its manipulative capabili enables a *direct* count without creating an intermediate file. Using the | symbol, it connects tw commands to create a *pipeline*:

```
$ ls | wc
     6       6      42                                  No filename this time!
```

See how you can arrive at the same result (except for the filename), this time by simply connectin the output of **`ls`** to the input of **`wc`**. No intermediate file is now needed. On seeing the symbol (known as a *pipe*), the shell performs the job of connecting the two commands. If you can conne a number of UNIX commands in this way, you can perform difficult tasks quite easily.

1.4.13 Programming with the Shell

The system also features a programming facility. You can assign a value to a variable at the prompt:

```
$ x=5                                  No spaces on either side of =
$ _
```

and then evaluate the value of this variable with the **echo** command and a **$**-prefixed variable name:

```
$ echo $x                              A $ required during evaluation
5
```

Apart from playing with variables, UNIX also provides control structures like conditionals and loops, and you'll see a great deal of that in later chapters.

1.4.14 exit: Signing Off

So far, what you have seen is only a small fragment of the UNIX giant, though you have already been exposed to some of its key features. Most of these commands will be considered in some detail in subsequent chapters, and it's a good idea to suspend the session for the time being. You should use the **exit** command to do that:

```
$ exit
login:
```

Alternatively, you may be able to use *[Ctrl-d]* (generated by pressing the *[Ctrl]* key and the character **d** on the keyboard) to quit the session. The `login:` message confirms that the session has been terminated, thus making it available for the next user.

Note: Depending on how your environment has been set up, you may not be able to use *[Ctrl-d]* to exit the session. If that happens, try the **logout** command, and if that fails too, use the **exit** command. This command will *always* work.

Caution: Make sure that you log out after your work is complete. If you don't do that, anybody can get hold of your terminal and continue working using your user-id. She may even remove your files! The login prompt signifies a terminated session, so don't leave your place of work until you see this prompt.

1.5 HOW IT ALL CLICKED

Until UNIX came on the scene, operating systems were designed with a particular machine in mind. They were invariably written in a low-level language (like assembler, which uses humanly unreadable code). The systems were fast but were restricted to the hardware they were designed for. Programs designed for one system simply wouldn't run on another. That was the status of the computer industry when Ken Thompson and Dennis Ritchie, of AT&T fame, authored the UNIX system.

General-Purpose Utilities

The best way to start acquiring knowledge of the UNIX command set is to try your hand at som of the general-purpose utilities of the system. These commands have diverse functions, but can b broadly divided into two categories. Some of them act as handy accessories that can perforn calculations or handle your mail, for instance. Others tell you the state the system is in, and eve manipulate it.

Every command featured in this chapter is useful, and has not been included here for cosmeti effect. Many of them have been reused in later chapters, especially in shell programming. A few ar true dark horses. You'll need these commands in all situations in your daily life at the machin The commands are simple to use, have very few options (except for **stty**), and don't require you t know much about the files they may access.

WHAT YOU WILL LEARN

- Display the calendar of a month or year with **cal**.
- Display the current system date and time in a variety of formats using **date**.
- Use **echo** with escape sequences to display a message on the terminal.
- Use **bc** as a calculator with a decimal, octal or hexadecimal number as base.
- The basics of electronic mail and its addressing scheme.
- Handle your mail with the character-based **mailx** program.
- Change your password with **passwd**.
- Display the list of users currently working on the system with **who**.
- Display the characteristics of your operating system with **uname**.
- Display the filename of your terminal with **tty**.
- Use **stty** to display and change your terminal's settings.

TOPICS OF SPECIAL INTEREST

- Features of the POSIX-recommended **printf** command as a portable replacement of **echo**.
- Record your login session including all keystrokes with **script**.

- The advantage character-based mailers have over graphic programs.
- The significance of the *mailbox* and *mbox* in the mailing system.

3.1 cal: THE CALENDAR

You can invoke the **cal** command to see the calendar of any specific month or a complete year. The facility is totally accurate and takes into account the leap year adjustments that took place in the year 1752. Let's have a look at its syntax drawn from the Solaris man page:

```
cal [ [ month ] year ]
```

Everything within rectangular brackets is optional, so we are told *(2.9.1)*. So, **cal** can be used without arguments, in which case it displays the calendar of the current month:

```
$ cal
      August 2005
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

The syntax also tells us that when **cal** is used with arguments, the month is optional but the year is not. To see the calendar for the month of March 2006, you need two arguments:

```
$ cal 03 2006
      March 2006
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

You can't hold the calendar of a year in a single screen page; it scrolls off too rapidly before you can use *[Ctrl-s]* to make it pause. To make **cal** pause in the same way **man** pauses, use **cal** with a pager (**more** or **less**) using the | symbol to connect them. A single argument to **cal** is interpreted as the year:

```
cal 2003 | more                                   Or use less instead of more
```

The | symbol connects two commands (in a pipeline) where **more** takes input from the **cal** command. We have used the same symbol in Chapter 1 to connect the **ls** and **wc** commands *(1.4.12)*. You can now scroll forward by pressing the spacebar, or move back using **b**.

3.2 date: DISPLAYING THE SYSTEM DATE

The UNIX system maintains an internal clock meant to run perpetually. When the system is shut down, a battery backup keeps the clock ticking. This clock actually stores the number of seconds elapsed since the **Epoch**; this is January 1, 1970. A 32-bit counter stores these seconds (except on 64-bit machines), and the counter will overflow sometime in 2038.

You can display the current date with the **date** command, which shows the date and time to the nearest second:

```
$ date
Wed Aug 31 16:22:40 IST 2005                    Time zone here is IST
```

The command can also be used with suitable format specifiers as arguments. Each format is preceded by the + symbol, followed by the % operator, and a single character describing the format. For instance, you can print only the month using the format +%m:

```
$ date +%m
08
```

or the month name:

```
$ date +%h
Aug
```

or you can combine them in one command:

```
$ date +"%h %m"
Aug 08
```

There are many other format specifiers, and the useful ones are listed below:

d—The day of the month (1 to 31).

y—The last two digits of the year.

H, M and S—The hour, minute and second, respectively.

D—The date in the format *mm/dd/yy*.

T—The time in the format *hh:mm:ss*.

When you use multiple format specifiers (as shown in the previous example), you must enclose them within quotes (single or double), and use a single + symbol before it.

Note: You can't change the date as an ordinary user, but the system administrator uses the same command with a different syntax to set the system date! This is discussed in Chapter 15.

3.3 echo: DISPLAYING A MESSAGE

We have used the **echo** command a number of times already in this text. This command is often used in shell scripts to display diagnostic messages on the terminal, or to issue prompts for taking user input. So far, we have used it in two ways:

- To display a message (like **echo Sun Solaris**).
- To evaluate shell variables (like **echo $SHELL**).

Originally, **echo** was an external command *(2.5)*, but today all shells have **echo** built-in. There are some differences in **echo**'s behavior across the shells, and most of these differences relate to the way **echo** interprets certain strings known as escape sequences.

An **escape sequence** is generally a two character-string beginning with a \ (backslash). For instance, \c is an escape sequence. When this escape sequence is placed at the end of a string used as an argument to **echo**, the command interprets the sequence as a directive to place the cursor and prompt in the same line that displays the output:

```
$ echo "Enter filename: \c"
Enter filename: $ _                    Prompt and cursor in same line
```

This is how **echo** is used in a shell script to accept input from the terminal. Like \c, there are other escape sequences (Table 3.1). Here are two commonly used ones:

\t—A tab which pushes text to the right by eight character positions.

\n—A newline which creates the effect of pressing *[Enter]*.

All escape sequences are not two-character strings. ASCII characters can also be represented by their octal values (numbers using the base 8 contrasted with the standard decimal system which uses the base 10). **echo** interprets a number as octal when it is preceded by **\0**. For instance, *[Ctrl-g]*, which results in the sounding of a beep, has the octal value 7 (i.e., \07). You can use this value as an argument to **echo**, but only after preceding it with \0:

```
$ echo '\07'                           Double quotes will also do
..... beep heard .....
```

This is the first time we see ASCII octal values used by a UNIX command. Later, you'll see the **tr**, **awk** and **perl** commands also using octal values.

Caution: **echo** escape sequences are a feature of System V. BSD doesn't recognize them but it supports the -n option as an alternative to the \c sequence:

```
echo "Enter filename: \c"              System V
echo -n "Enter filename: "             BSD
```

Even though we don't use the disk version nowadays, the bad news is that the shells too respond in different ways to these escape sequences. Rather than go into these details, a word of caution from POSIX would be appropriate: Use **printf**.

LINUX: Bash, the standard shell used in Linux, interprets the escape sequences only when **echo** is used with the -e option:

```
echo -e "Enter your name:\c"
```

We'll be using these escape sequences extensively in this text, so if you are a Bash user (which most Linux users are), you must either commit this option to memory or make a special setting in the shell that makes **echo** behave in the normal way *(14.1—Tip)*.

Table 3.1 Escape Sequences Used by **echo** and **printf**

Escape Sequence	*Significance*
`\a`	Bell
`\b`	Backspace
`\c`	No newline (cursor in same line)
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\0`*n*	ASCII character represented by the octal value *n*, where *n* can't exceed 0377 (decimal value 255)

3.4 printf: AN ALTERNATIVE TO echo

The **printf** command is available on most modern UNIX systems, and is the one you should use instead of **echo** (unless you have to maintain a lot of legacy code that use **echo**). Like **echo**, it exists as an external command; it's only the Bash shell that has **printf** built-in. The command in its simplest form can be used in the same way as **echo**:

```
$ printf "No filename entered\n"                    \n is explicitly specified
No filename entered
$ _
```

printf also accepts all escape sequences used by **echo**, but unlike **echo**, it doesn't automatically insert a newline unless the `\n` is used explicitly. Though we don't need to use quotes in this example, it's good discipline to use them. **printf** also uses formatted strings in the same way the C language function of the same name uses them:

```
$ printf "My current shell is %s\n" $SHELL          No comma before $
My current shell is /usr/bin/bash
```

The `%s` format string acts as a placeholder for the value of `$SHELL` (the argument), and **printf** replaces `%s` with the value of `$SHELL`. `%s` is the standard format used for printing strings. **printf** uses many of the formats used by C's **printf** function. Here are some of the commonly used ones:

`%s` — String

`%30s` — As above but printed in a space 30 characters wide

`%d` — Decimal integer

`%6d` — As above but printed in a space 6 characters wide

`%o` — Octal integer

`%x` — Hexadecimal integer

`%f` — Floating point number

Note that the formats also optionally use a number to specify the width that should be used when printing a string or number. You can also use multiple formats in a single **printf** command. But then you'll have to specify as many arguments as there are format strings—and in the right order.

While **printf** can do everything that **echo** does, some of its format strings can convert data from one form to another. Here's how the number 255 is interpreted in octal (base 8) and hexadecimal (base 16):

```
$ printf "The value of 255 is %o in octal and %x in hexadecimal\n" 255 255
The value of 255 is 377 in octal and ff in hexadecimal
```

The %o and %x format strings are also used by **awk** and **perl** (and by C) to convert a decimal integer to octal and hex, respectively; it's good to know them. Note that we specified 255 twice to represent the two arguments because it's the same number that we want to convert to octal and hex.

Note: C language users should note some syntactical differences in **printf** usage. **printf** is a function in C and hence uses the parentheses to enclose its arguments. Moreover, arguments are separated from one another as well as from the format string by commas. Here's how the previous command line is implemented as a C statement:

```
printf("The value of 255 is %o in octal and %x in hexadecimal\n", 255, 255);
```

The discussion on **printf** should prepare you well for eventually using the **printf** function in C, but remember that this C function uses many format specifiers not used by the UNIX **printf** command.

3.5 bc: THE CALCULATOR

UNIX provides two types of calculators—a graphical object (the **xcalc** command) that looks like one, and the text-based **bc** command. The former is available in the X Window system and is quite easy to use. The other one is less friendly, extremely powerful and remains one of the system's neglected tools.

When you invoke **bc** without arguments, the cursor keeps on blinking and nothing seems to happen. **bc** belongs to a family of commands (called *filters*) that expect input from the keyboard when used without an argument. Key in the following arithmetic expression and then use *[Ctrl-d]* to quit **bc**:

```
$ bc
12 + 5
17                                        Value displayed after computation
[Ctrl-d]                                  The eof character
$ _
```

bc shows the output of the computation in the next line. Start **bc** again and then make multiple calculations in the same line, using the **;** as delimiter. The output of each computation is, however, shown in a separate line:

```
12*12 ; 2^32                              ^ indicates "to the power of"
144
4294967296                                Maximum memory possible on a 32-bit machine
```

bc performs only integer computation and truncates the decimal portion that it sees. This shows up clearly when you divide two numbers:

```
9/5
1                                        Decimal portion truncated
```

To enable floating-point computation, you have to set **scale** to the number of digits of precision before you key in the expression:

```
scale=2                                  Truncates to 2 decimal places
17/7
2.42                                     Not rounded off, result is actually 2.42857.....
```

bc is quite useful in converting numbers from one base to another. For instance, when setting IP addresses (*17.1.3*) in a network, you may need to convert binary numbers to decimal. Set **ibase** (input base) to 2 before you provide the number:

```
ibase=2
11001010                                 Output in decimal—base 10
202
```

The reverse is also possible, this time with **obase**:

```
obase=2
14                                       Binary of 14
1110
```

In this way, you can convert from one base to the other (not exceeding 16). **bc** also comes with a library for performing scientific calculations. It can handle very, very large numbers. If a computation results in a 900-digit number, **bc** will show each and every digit!

3.6 script: RECORDING YOUR SESSION

This command, virtually unknown to many UNIX users, lets you "record" your login session in a file. This command is not included in POSIX, but you'll find it useful to store in a file all keystrokes as well as output and error messages. You can later view the file. If you are doing some important work and wish to keep a log of all your activities, you should invoke this command immediately after you log in:

```
$ script
Script started, file is typescript
$ _                                      Another shell—child of login shell
```

The prompt returns and all your keystrokes (including the one used to backspace) that you now enter here get recorded in the file **typescript**. After your recording is over, you can terminate the session by entering **exit**:

```
$ exit                                   Or use [Ctrl-d]
Script done, file is typescript
$ _                                      Back to login shell
```

- Use a mix of alphabetic or numeric characters. Enterprise UNIX systems won't allow passwords that are wholly alphabetic or numeric.
- Don't write down the password in an easily accessible document.
- Change the password regularly.

You must remember your password, but if you still forget it, rush to your system administrator. You'll learn later of the terrible consequences that you may have to face if people with mischievous intent somehow come to know what your password is. The command also behaves differently when used by the system administrator; it doesn't ask for the old password. The **passwd** command is revisited in Chapter 15.

3.10 who: WHO ARE THE USERS?

UNIX maintains an account of all users who are logged on to the system. It's often a good idea to know their user-ids so you can mail them messages. The **who** command displays an informative listing of these users:

```
$ who
root       console     Aug  1 07:51    (:0)
kumar      pts/10      Aug  1 07:56    (pc123.heavens.com)
sharma     pts/6       Aug  1 02:10    (pc125.heavens.com)
project    pts/8       Aug  1 02:16    (pc125.heavens.com)
sachin     pts/14      Aug  1 08:36    (mercury.heavens.com)
```

The first column shows the usernames (or user-ids) of five users currently working on the system. The second column shows the device names of their respective terminals. These are actually the filenames associated with the terminals. kumar's terminal has the name pts/10 (a file named 10 in the pts directory). The third, fourth and fifth columns show the date and time of logging in. The last column shows the machine name from where the user logged in. Users can log in remotely to a UNIX system, and all users here except root have logged in remotely from four different machines.

While it's a general feature of most UNIX commands to avoid cluttering the display with header information, this command does have a header option (-H). This option prints the column headers, and when combined with the -u option, provides a more detailed list:

```
$ who -Hu
NAME       LINE        TIME           IDLE    PID  COMMENTS
root       console     Aug  1 07:51   0:48  11040     (:0)
kumar      pts/10      Aug  1 07:56   0:33  11200     (pc123.heavens.com)
sachin     pts/14      Aug  1 08:36    .    13678     (mercury.heavens.com)
```

Two users have logged out, so it seems. The first five columns are the same as before, but it's the sixth one (IDLE) that is interesting. A . against sachin shows that activity has occurred in the last one minute before the command was invoked. kumar seems to be idling for the last 33 minutes. The PID is the process-id, a number that uniquely identifies a process. You have seen it when you used the **ps** command in Section 1.4.8 to list the processes running at your terminal.

One of the users shown in the first column is obviously the user who invoked the **who** command. To know that specifically, use the arguments am and i with **who**:

```
$ who am i
kumar        pts/10        Aug  1 07:56    (pc123.heavens.com)
```

Note: UNIX provides a number of tools (called *filters*) to extract data from command output for further processing. For instance, you can use the **cut** command to extract the first column from the who output, and then use this list with **mailx** to send a message to these users. The ability to combine commands to perform tasks that are not possible to achieve using a single command is what makes UNIX so different from other operating systems. We'll be combining commands several times in this text.

3.11 uname: KNOWING YOUR MACHINE'S CHARACTERISTICS

The **uname** command displays certain features of the operating system running on your machine. By default, it simply displays the name of the operating system:

```
$ uname
SunOS                                          Linux shows Linux
```

This is the operating system used by Sun Solaris. Linux systems simply show the name Linux. Using suitable options, you can display certain key features of the operating system, and also the name of the machine. The output will depend on the system you are using.

The Current Release (-r) Since UNIX comes in many flavors, vendors have customized a number of commands to behave in the way they want, and not as AT&T decreed. A UNIX command often varies across versions so much so that you'll need to use the -r option to find out the version of your operating system:

```
$ uname -r
5.8                                            This is SunOS 5.8
```

This is a machine running SunOS 5.8, the name given to the operating system used by the Solaris 8 environment. If a command doesn't work properly, it could either belong to a different "implementation" (could be BSD) or a different "release" (may be 4.0, i.e., System V Release 4 of AT&T).

The Machine Name (-n) If your machine is connected to a network, it must have a name (called *hostname*). If your network is connected to the Internet, then this hostname is a component of your machine's *domain name* (a series of words separated by dots, like *mercury.heavens.com*). The -n option tells you the hostname:

```
$ uname -n
mercury                                        The first word of the domain name
```

The same output would be obtained with the **hostname** command. Many UNIX networking utilities use the hostname as argument. To copy files from a remote machine named *mercury* running the FTP service, you have to run **ftp mercury**.

LINUX: **uname -n** may show either the host name (like `mercury`) or the complete domain name (like `mercury.heavens.com`), depending on the flavor of Linux you are using. **uname** and **uname -r** display the operating system name and version number of the kernel, respectively:

```
$ uname
Linux
$ uname -r
2.4.18-14                                                    Kernel version is 2.4
```

The first two numbers of the kernel version (here, 2.4) are something every Linux user *must remember*. Before installing software, the documentation may require you to use a kernel that is "at least" 2.2. The same software should run on this machine whose kernel version is 2.4.

3.12 tty: KNOWING YOUR TERMINAL

Since UNIX treats even terminals as files, it's reasonable to expect a command that tells you the filename of the terminal you are using. It's the **tty** (teletype) command, an obvious reference to the device that has now become obsolete. The command is simple and needs no arguments:

```
$ tty
/dev/pts/10
```

The terminal filename is 10 (a file named 10) resident in the pts directory. This directory in turn is under the /dev directory. These terminal names were seen on a Solaris machine; your terminal names could be different (say, `/dev/tty01`).

You can use **tty** in a shell script to control the behavior of the script depending on the terminal it is invoked from. If a program must run from only one specified terminal, the script logic must use **tty** to make this decision.

3.13 stty: DISPLAYING AND SETTING TERMINAL CHARACTERISTICS

Different terminals have different characteristics, and your terminal may not behave in the way you expect it to. For instance, command interruption may not be possible with *[Ctrl-c]* on your system. Sometimes you may like to change the settings to match the ones used at your previous place of work. The **stty** command helps straighten these things out; it both displays and changes settings.

stty uses a very large number of *keywords* (options that look different), but we'll consider only a handful of them. The -a (all) option displays the current settings. A trimmed output is presented below:

```
$ stty -a
speed 38400 baud; rows = 25; columns = 80; ypixels = 0; xpixels = 0;
intr = ^c; quit = ^\; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -defecho -flusho -pendin iexten
```

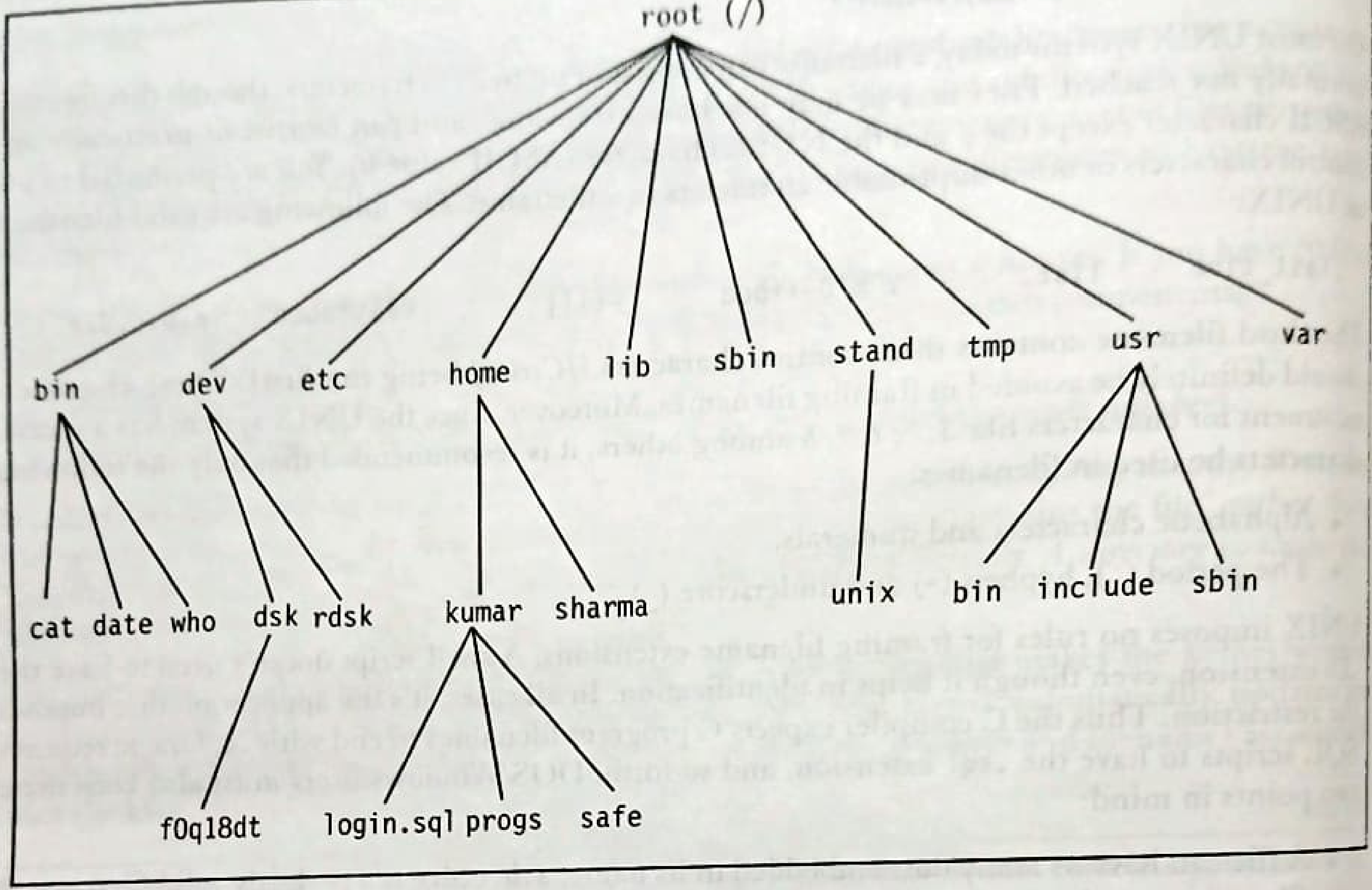

Fig. 4.1 The UNIX File System Tree

The root directory (/) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and other files under them. For instance, bin and usr are two directories directly under /, while a second bin and kumar are subdirectories under usr.

Every file, apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root. Thus, the home directory is the parent of kumar, while / is the parent of home, and the grandparent of kumar. If you create a file `login.sql` under the kumar directory, then kumar will be the parent of this file.

It's also obvious that, in these parent—child relationships, the parent is always a directory. home and kumar are both directories as they are both parents of at least one file or directory. `login.sql` is simply an ordinary file; it can't have any directory under it.

4.4 THE HOME VARIABLE: THE HOME DIRECTORY

When you log on to the system, UNIX automatically places you in a directory called the **home directory**. It is created by the system when a user account is opened. If you log in using the login name kumar, you'll land up in a directory that could have the pathname /home/kumar (or something else). You can change your home directory when you like, but you can also effect a quick return to it, as you'll see soon.

The shell variable HOME knows your home directory:

```
$ echo $HOME
/home/kumar                              First / represents the root directory
```

What you see above is an **absolute pathname**, which is simply a sequence of directory names separated by slashes. An absolute pathname shows a file's location with reference to the top, i.e., root. These slashes act as delimiters to file and directory names, except that *the first slash is a synonym for root*. The directory kumar is placed two levels below root.

It's often convenient to refer to a file foo located in your home directory as $HOME/foo. Further, most shells (except Bourne) also use the ~ symbol for this purpose. So, $HOME/foo is the same as ~/foo in these shells. The ~ symbol is a little tricky to use because it can refer to any user's home directory and not just your own. If user sharma has the file foo in his home directory, then kumar can access it as ~sharma/foo. The principle is this:

A tilde followed by / (like ~/foo) refers to one's own home directory, but when followed by a string (~sharma) refers to the home directory of that user represented by the string.

Note: The home directory is determined by the system administrator at the time of opening a user account. Its pathname is stored in the file /etc/passwd. On many UNIX systems, home directories are maintained under /home, but your home directory could be different (say, in /export/home). Even if you have moved away from your "home", you can use the cd command to effect a quick return to it, as you'll see soon.

4.5 pwd: CHECKING YOUR CURRENT DIRECTORY

UNIX encourages you to believe that, like a file, a user is *placed* in a specific directory of the file system on logging in. You can move around from one directory to another, but at any point of time, you are located in only one directory. This directory is known as your current directory.

At any time, you should be able to know what your current directory is. The **pwd** (print working directory) command tells you that:

```
$ pwd
/home/kumar
```

Like HOME, **pwd** displays the absolute pathname. As you navigate the file system with the **cd** command, you'll be using **pwd** to know your current directory.

Note: It's customary to refer to a file foo located in the home directory as $HOME/foo. Depending on the shell you use, it may be possible to even access foo as ~/foo. One form is shell-dependent but the other isn't, but both naming conventions are followed in this text.

4.6 cd: CHANGING THE CURRENT DIRECTORY

You can move around in the file system by using the **cd** (change directory) command. When used with an argument, it changes the current directory to the directory specified as argument, for instance, progs:

```
$ pwd
/home/kumar
$ cd progs                                        progs must be in current directory
$ pwd
/home/kumar/progs
```

Though **pwd** displays the absolute pathname, **cd** doesn't need to use one. The command **cd prog** here means this: "Change your subdirectory to progs under the current directory." Using a pathnam causes no harm either; use **cd /home/kumar/progs** for the same effect.

When you need to switch to the /bin directory where most of the commonly used UNIX command are kept, you should use the absolute pathname:

```
$ pwd
/home/kumar/progs
$ cd /bin                                         Absolute pathname required here becaus
$ pwd                                             bin isn't in current directory
/bin
```

We can also navigate to /bin (or any directory) using a different type of pathname; we are comin to that shortly.

cd can also be used without any arguments:

```
$ pwd
/home/kumar/progs
$ cd                                              cd used without arguments
$ pwd                                             reverts to the home directory
/home/kumar
```

Attention, DOS users! This command invoked without an argument doesn't indicate the curren directory. It simply switches to the home directory, i.e., the directory where the user originall logged into. Therefore, if you wander around in the file system, you can force an immediate retur to your home directory by simply using **cd**:

```
$ cd /home/sharma
$ pwd
/home/sharma
$ cd
$ pwd                                             Returns to home directory
/home/kumar
```

The **cd** command can sometimes fail if you don't have proper permissions to access the directo This doesn't normally happen unless you deliberately tamper with the permissions of the directo The technique of doing that is described in Section 6.5.

Note: Unlike in DOS, when **cd** is invoked without arguments, it simply reverts to its home directory. I doesn't show you the current directory!

4.7 mkdir: MAKING DIRECTORIES

Directories are created with the **mkdir** (make directory) command. The command is followed by names of the directories to be created. A directory patch is created under the current directory like this:

```
mkdir patch
```

You can create a number of subdirectories with one **mkdir** command:

```
mkdir patch dbs doc                                  Three directories created
```

So far, simple enough, but the UNIX system goes further and lets you create directory trees with just one invocation of the command. For instance, the following command creates a directory tree:

```
mkdir pis pis/progs pis/data                         Creates the directory tree
```

This creates three subdirectories—pis and two subdirectories under pis. The order of specifying the arguments is important; you obviously can't create a subdirectory before creation of its parent directory. For instance, you can't enter

```
$ mkdir pis/data pis/progs pis
mkdir: Failed to make directory "pis/data"; No such file or directory
mkdir: Failed to make directory "pis/progs"; No such file or directory
```

Note that even though the system failed to create the two subdirectories, progs and data, it has still created the pis directory.

Sometimes, the system refuses to create a directory:

```
$ mkdir test
mkdir: Failed to make directory "test"; Permission denied
```

This can happen due to these reasons: * ✓

- The directory test may already exist.
- There may be an ordinary file by that name in the current directory.
- The permissions set for the current directory don't permit the creation of files and directories by the user. You'll most certainly get this message if you try to create a directory in /bin, /etc or any other directory that houses the UNIX system's files.

We'll take up file and directory permissions in Chapter 6 featuring file attributes.

4.8 rmdir: REMOVING DIRECTORIES

The **rmdir** (remove directory) command removes directories. You simply have to do this to remove the directory pis:

```
rmdir pis                                            Directory must be empty
```

Like **mkdir**, **rmdir** can also delete more than one directory in one shot. For instance, the three directories and subdirectories that were just created with **mkdir** can be removed by using **rmdir** with a reversed set of arguments:

```
rmdir pis/data pis/progs pis
```

Note that when you delete a directory and its subdirectories, a reverse logic has to be applied. The following directory sequence used by **mkdir** is invalid in **rmdir**:

```
$ rmdir pis pis/progs pis/data
rmdir: directory "pis": Directory not empty
```

Have you observed one thing from the error message? **rmdir** has silently deleted the lowest level subdirectories progs and data. This error message leads to two important rules that you should remember when deleting directories:

- You can't delete a directory *with* **rmdir** unless it is empty. In this case, the pis directory couldn't be removed because of the existence of the subdirectories, progs and data, under it.
- You can't remove a subdirectory unless you are placed in a directory which is hierarchically *above* the one you have chosen to remove.

The first rule follows logically from the example above, but the highlight on **rmdir** has significance that will be explained later. (A directory can also be removed without using **rmdir**.) To illustrate the second cardinal rule, try removing the progs directory by executing the command from the same directory itself:

```
$ cd progs
$ pwd
/home/kumar/pis/progs
$ rmdir /home/kumar/pis/progs                    Trying to remove the current directory
rmdir: directory "/home/kumar/pis/progs": Directory does not exist
```

To remove this directory, you must position yourself in the directory above progs, i.e., pis, and then remove it from there:

```
$ cd /home/kumar/pis
$ pwd
/home/kumar/pis
$ rmdir progs
```

The **mkdir** and **rmdir** commands work only in directories *owned* by the user. Generally, a user is the owner of her home directory, and she can create and remove subdirectories (as well as regular files) in this directory or in any subdirectories created by her. However, she normally won't be able to create or remove files and directories in other users' directories. The concept of ownership will be discussed in Section 6.3.

Note: A subdirectory can't be removed with **rmdir** unless it's empty, and one is positioned in its parent directory or above it. But we can remove a directory without using **rmdir** also (discussed later).

HOW IT WORKS: How Files and Directories are Created and Removed

As mentioned in Section 4.1.2, a file (ordinary or directory) is associated with a name and a number, called the *inode number*. When a directory is created, an entry comprising these two parameters is made in the file's parent directory. The entry is removed when the directory is removed. The same holds good for ordinary files also. Figure 4.2 highlights the effect of **`mkdir`** and **`rmdir`** when creating and removing the subdirectory `progs` in `/home/kumar`.

Filename	*Inode Number*
.	386444
..	417585
foo	499770

`mkdir progs` →

Filename	*Inode Number*
.	386444
..	417585
foo	499770
progs	162112

`rmdir progs` →

Filename	*Inode Number*
.	386444
..	417585
foo	499770

Fig. 4.2 Directory Entry after **`mkdir`** and **`rmdir`**

Later in this chapter, we'll discuss the significance of the entries, `.` and `..`, that you'll find in every directory. In this chapter and Chapters 5 and 11, we'll be progressively monitoring this directory for changes that are caused by some of the file-handling commands.

4.9 ABSOLUTE PATHNAMES

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory. For instance, the command

```
cat login.sql
```

will work only if the file `login.sql` exists in your current directory. However, if you are placed in `/usr` and want to access `login.sql` in `/home/kumar`, you can't obviously use the above command, but rather the pathname of the file:

```
cat /home/kumar/login.sql
```

As stated before, if the first character of a pathname is /, the file's location must be determined with respect to root (the first /). Such a pathname, as the one above, is called an **absolute pathname**. When you have more than one / in a pathname, for each such /, you have to descend one level in the file system. Thus, `kumar` is one level below `home`, and two levels below root.

When you specify a file by using frontslashes to demarcate the various levels, you have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. You can have two files with the same name, but in different directories; their pathnames will also be different. Thus, the file `/home/kumar/progs/c2f.pl` can coexist with the file `/home/kumar/safe/c2f.pl`.

4.9.1 Using the Absolute Pathname for a Command

More often than not, a UNIX command runs by executing its disk file. When you specify the **date** command, the system has to locate the file date from a list of directories specified in the PATH variable, and then execute it. However, if you know the location of a particular command, you can precede its name with the complete path. Since **date** resides in /bin (or /usr/bin), you can also use the absolute pathname:

```
$ /bin/date
Thu Sep  1 09:30:49 IST 2005
```

Nobody runs the **date** command like that. For any command that resides in the directories specified in the PATH variable, you don't need to use the absolute pathname. This PATH, you'll recall *(2.4.1)*, invariably has the directories /bin and /usr/bin in its list.

If you execute programs residing in some other directory that isn't in PATH, the absolute pathname then needs to be specified. For example, to execute the program **less** residing in /usr/local/bin, you need to enter the absolute pathname:

```
/usr/local/bin/less
```

If you are frequently accessing programs in a certain directory, it's better to include the directory itself in PATH. The technique of doing that is shown in Section 10.3.

4.10 RELATIVE PATHNAMES

You would have noted that in a previous example *(4.8)*, we didn't use an absolute pathname to move to the directory progs. Nor did we use one as an argument to **cat** *(4.9)*:

```
cd progs
cat login.sql
```

Here, both progs and login.sql are presumed to exist in the current directory. Now, if progs also contains a directory scripts under it, you still won't need an absolute pathname to change to that directory:

```
cd progs/scripts                                    progs is in current directory
```

Here we have a pathname that has a /, but it is not an absolute pathname because it doesn't begin with a /. In these three examples, we used a rudimentary form of relative pathnames though they are generally not labeled as such. Relative pathnames, in the sense they are known, are discussed next.

4.10.1 Using . and .. in Relative Pathnames

In a preceding example *(4.8)*, you changed your directory from /home/kumar/pis/progs to its parent directory (/home/kumar/pis) by using **cd** with an absolute pathname:

```
cd /home/kumar/pis
```

Navigation often becomes easier by using a common ancestor (here, /home) as reference. UNIX offers a shortcut—the **relative pathname**—that uses either the current or parent directory as reference, and specifies the path relative to it. A relative pathname uses one of these cryptic symbols

- . (a single dot)—This represents the current directory.
- .. (two dots)—This represents the parent directory.

We'll now use the .. to frame relative pathnames. Assuming that you are placed in /home/kumar/progs/data/text, you can use .. as an argument to **cd** to move to the parent directory, /home/kumar/progs/data:

```
$ pwd
/home/kumar/progs/data/text
$ cd ..                                           Moves one level up
$ pwd
/home/kumar/progs/data
```

This method is compact and more useful when ascending the hierarchy. The command **cd ..** translates to this: "Change your directory to the parent of the current directory." You can combine any number of such sets of .. separated by /s. However, when a / is used with .. it acquires a different meaning; instead of moving down a level, it moves one level *up*. For instance, to move to /home, you can always use **cd /home**. Alternatively, you can also use a relative pathname:

```
$ pwd
/home/kumar/pis
$ cd ../..                                        Moves two levels up
$ pwd
/home
```

Now let's turn to the solitary dot that refers to the current directory. Any command which uses the current directory as argument can also work with a single dot. This means that the **cp** command *(5.2)* which also uses a directory as the last argument can be used with a dot:

```
cp ../sharma/.profile .                          A filename can begin with a dot
```

This copies the file .profile to the current directory (.). Note that you didn't have to specify the filename of the copy; it's the same as the original one. This dot is also implicitly included whenever we use a filename as argument, rather than a pathname. For instance, **cd progs** is the same as **cd ./progs**.

Note: Absolute pathnames can get very long if you are located a number of "generations" away from root. However, whether you should use one depends solely on the number of keystrokes required when compared to a relative pathname. In every case here, the relative pathname required fewer key depressions. Depending on where you are currently placed, an absolute pathname can be faster to type.